Specification-based Verification of Incomplete Programs

Le V. Tien[1], Quan T. Tho[2], and Le D. Anh[3]
[1] Faculty of Computer Science and Engineering, Hochiminh City University of Technology, Hochiminh, Vietnam
Email: tienlv@younetco.com
[2] Faculty of Computer Science and Engineering, Hochiminh City University of Technology, Hochiminh, Vietnam
Email: qttho@cse.hcmut.edu.vn
[3] Faculty of Computer Science and Engineering, Hochiminh City University of Technology, Hochiminh, Vietnam
Email: tintinkool@gmail.com

***Abstract*—Recently, formal methods like model checking or theorem proving have been considered efficient tools for software verification. However, when practically applied, those techniques suffer high complexity cost. Combining static analysis with dynamic checking to deal with this problem has been becoming an emerging trend, which results in the introduction of concolic testing technique and its variations. However, the analysis-based verification techniques always assume the availability of full source code of the verified program, which does not always hold in real life contexts. In this paper, we propose an approach to tackle this problem, where our contributed ideas are (i) combining function specification with control flow analysis to deal with source-missing function; (ii) generating self-complete programs from incomplete programs by means of concrete execution, thus making them fully verifiable by model checking; and (iii) developing a constraint-based test-case generation technique to significantly reduce the complexity. Our solution has been proved viable when successfully deployed for checking programming work of students.**

***Index Terms*—specification-based model checking, concolic testing, constraint-based test-case generation, incomplete programs**

I. Introduction

Recently, formal methods, e.g. model checking [1] or theorem proving [2], have been increasingly applied for software verification. Whereas those techniques were proved efficient, at least theoretically, to identify and explain real bugs in a program, they suffer state explosion problem even with some simple typical non-trivial cases of verification. To tackle this problem, one of remarkable approaches is to adopt test-case generation techniques used in software testing to produce only sufficient input when performing model-based verification of programs. Especially, the *concolic testing* technique, which is a hybrid software verification technique that interleaves concrete execution (testing on particular inputs) with symbolic execution [7] has been emerging as a efficient test-case generation technique. Recently, DART [3], SYNERGY [4], DUALIZER [5] and DASH [6] are notable approaches based on this idea.

However, those techniques require that the whole source code of the verified program must be available for analysis. This assumption does not always hold in practical situations of software development context; particularly when the program invokes some library functions provided as binary form.

In this paper, this situation is regarded as the *verification of incomplete program*[1], which we propose a framework to tackle. By doing so, we have made the following key research contributions:

- Handling the problem of analyzing source-missing functions by combining function specification with program control-flow to produce *combined constraints* sufficiently covering all of possible scenarios.
- Using concrete execution to replace functions invocation by the generated output value. Thus, the incomplete programs will be transformed into self-complete program fully available for model checking.
- An algorithm known as *CTG*^{E} (Efficient Constraint-based Test-case Generation) to generate combined constraints in linear time, instead of exponential time suffered by the brute-force approach.

The rest of the paper is organized as follows. Section II gives some relevant background. Section III discusses some motivating examples. In Section IV, we present our general verification framework. The CTG^E algorithm is presented in Section V. The next sections give some experimental results and conclude the paper.

II. Background: Model Checking and The Concolic Testing Techniques

A. Model Checking

Model checking, first termed by Clarke and Emerson [1], is an automatic verification technique for finite state concurrent systems. In model checking, the system/program to be verified is first formalized as a mathematical model. In model checking, the model is often in the form of Kripke structure [9]. Basically, a Kripke structure can be considered as a nondeterministic finite automaton in which the temporal logic [10] can be applied to verify a certain characteristic of an input string.

Compared to other verification techniques, model checking offers a practically useful advantage of producing counter-example when detecting system error. Such, the error can be

[1] The term incomplete here implies the lack of some parts in the source code, not the completeness of program in terms of functionality.

DOI: 01.IJIT.02.02.46

inspected in an obvious and convincing manner. There are many attempts made to improve the capability of generating counter-examples for model checking, ranging from symbolic approach [11] to probabilistic approach [12][13]. They can be introduced as a general framework [14] or a work aiming to a specific model checking software [15].

B. The Concolic Testing Techniques

LISTING I. AN ILLUSTRATED EXAMPLE OF CONCOLIC TESTING

```
void foo (int x, int y)
{
0: if (x != y)
1:    if (2*x = x + 10)
2:       error();
}
```

The concolic testing, which is a hybrid software verification technique combines concrete execution with symbolic execution [7], has emerged recently as an efficient technique for test-case generation. As compared to traditional white-box testing, the concolic technique attracts much attention due to its capability of reducing the number of path conditions to be explored.

For example, with the program given in Listing I, there are two branch conditions of (x!=y) and (2*x = x+10). For traditional white-box testing, there would be three path conditions needed to be considered. When concolic testing is applied, it first randomizes arbitrary values for x and y, e.g. x=1 and y=2. In the concrete execution, the test in line 0 is reached since the condition of x!=y is true but the test in line 1 failed because the condition 2*x = x+10 is false. Concurrently, the symbolic execution follows the same path but treating x and y as symbolic variables. The condition of (x!=y) $\wedge$ (2*x != x+10) now is called a *path conditions*. To let the verification follow a different execution path on the next run, this approach takes the last path condition encountered, i.e. 2*x != x+10, and then negates it, producing 2*x = x+10. An automated theorem prover is then invoked to find values for the input variables x and y satisfying the new produced condition x!=y $\wedge$ 2*x=x+10. Let them be x=10, y=5, for instance. Running the program on this input set reaches the error. Thus, we only need to explore 2 path conditions if using concolic testing.

III. MOTIVATING EXAMPLE: MODEL CHECKING ON INCOMPLETE PROGRAM

A. Incomplete program

To give a clear illustration on our motivation of this research, let us consider the following verification context of modular program, as presented in Listing II. In the original program, given in Listing II(a), the main function will subsequently call the two functions *func1* and *func2*. Among them, *func1* is a simple library function and *func2* is doing a critical task which is needed to be verified carefully. It is assumed that if the value *T-error* is passed to *func2*, the simulated model checking can detect a real bug of a program accordingly.

LISTING II. TRANSFORMING A MODULAR PROGRAM FOR MODEL CHECKING

```
int func1(int n)
{
   if (n>10) return 2*n;
   else if (n<5) return T-error;
   else return n;
}

void func2 (int b)
{
   //doing some critical tasks T
   // error will be detect if the input is T-error
   T;
}

void main ()
{
   int a=1,b=1;
   read_from_input(&a);
   if (a>0) b = func1(a);
   func2(b);
   return OK;
}
```

(a) Modular program

```
void main ()
{
   int a=1,b-1;
   read_from_input(&a);
   if (a>0) {
      if (a>10) b = 2*a;
      else if (a<5) return T-error;
      else b = a;
   }
   T'(b);
   return OK;
}
```

(b) Transformed program

```
/*@ requires  n>0
  @ ensures (n>10 => \result==2*n) || (n<5 => \result==T_error) ||
((n<=10 && n>=5) => \result==n)
  */
int func1(int n);

void main ()
{
   int a=1,b=1;
   read_from_input(&a);
   if (a>0) b = func1(a);
   func2(b);
   return OK;
}
```

(c) Incomplete program

Typically, in order to check this program using model checking technique, one needs to transform the program into a non-modular form similar to that which is given in Listing II(b). When using concolic approach on the transformed program, it is easily observable that there are 4 path constraints generated of (i) a>10 (ii) a<=10 $\wedge$ a>5; (iii) a<5 $\wedge$ a>0; and (iv) a<0. Solving constraint (iii) will give us necessary test-case leading to the error, e.g. a=3; In Listing II(b), *T'* implies the transformed code of *T*.

However, in order to perform the transformation as discussed, one would need the full source code of *func1*. It is not always possible in practical situations, where *func1* may be a function commonly used from an existing library. In other

DOI: 01.IJIT.02.02.46

words, one may need to verify a program without its full source code. In this paper, this situation is regarded as the verification of an *incomplete program*.

As illustrated in Listing II(c), we do not possess the source code *func1*, supposedly called from a dynamic library. This will pose two major problems when one wants to verify this program: (i) lack of the transformed code to be further processed by a model checking tool; and (ii) lack of enough test-case to be verified sufficiently when providing input for the corresponding model.

For clearer observation of the problem (ii), let us consider using concolic testing for the program in Listing II(c). There will be two test-cases to be generated, corresponding to the two constraints of $a>0$ and $a<0$. Obviously, there is a risk that if the two concrete test-cases are $a=17$ and $a=-2$, the potential error when *func1* probably returns *T-error* will not be detected.

B. Generation of combined constraint and simulated results

Here, our approach to overcome this problem is that we may assume all library functions are well-defined, i.e. their semantics can be annotated using pre-conditions and post-conditions as depicted in Listing II(c). By combining of the pre/post-condition and with the path constraints of the program, one can obtain sufficient combined constraints for generated test-case. For example, in Listing II(c), when analyzing the path constraints, one will obtain (τ_1): $a>0$ and (τ_2): $a<=0$. By analyzing the pre/post-conditions, the following addition constraints are added: (ρ_1): $a>10$; (ρ_2): $a<=10 \wedge a>5$; and (ρ_3): $a<=5 \wedge$

LISTING III. GENERATED SELF-COMPLETE PROGRAMS FOR MODEL CHECKING

```
void main ()
{
  int a=1,b=1;
  a = 12; //testcase 1
  if (a>0) b = 24;
  T'(b);
  return OK;
}
```
(a) Simulated program with testcase a = 12

```
void main ()
{
  int a=1,b=1;
  a = 7; //testcase 2
  if (a>0) b = 7;
  T'(b);
  return OK;
}
```
(b) Simulated program with testcase a = 7

```
void main ()
{
  int a=1,b=1;
  a = 3; //testcase 3
  if (a>0) b = T-error;
  T'(b);
  return OK;
}
```
(c) Simulated program with testcase a = 3

```
void main ()
{
  int a=1,b=1;
  a = 7; //testcase 4
  if (a>0) b = unknown;
  T'(b);
  return OK;
}
```
(d) Simulated program with testcase a = 7

$a>0$[1]. Then, we generate the following valid combined constraints of ($\tau_1 \wedge \rho_1$): $a>10$; ($\tau_1 \wedge \rho_2$): $a<=10 \wedge a>5$; ($\tau_1 \wedge \rho_3$): $a<=5 \wedge a>0$ and (τ_2): $a<=0$. Using a solver to solve those constraints, sufficient test-case will be obtained;
e.g. $a=12$, $a=7$, $a=3$ and $a=-4$[3].

In the next-step, to tackle the source-missing problem of *func1*, this function is invoked with the specific test-case generated above. As a result, we will acquire the actual outputs, which are respectively 24, 7 and *T-error*. The test-case of $a=-4$ will not be used since a symbolic execution may point out that this test-case cannot pass through the checking condition to reach the function call.

Next, instead of transforming the source code, the result of *func1* is simulated using the generated output just obtained about. As a result, there are four newly generated programs whose source codes are self-complete as presented in Listing III. Hence, we can identify the bug when processing the one generated in Listing III(c).

IV. SPECIFICATION-BASED MODEL CHECKING FRAMEWORK

The verification framework is proposed in Fig. 1. As discussed in the motivating example, the verification on incomplete programs consists of the following major steps:

- *Specification-based Test-case Generation*: It generates test-case based on the combination of constraints inferred from function specification with those from control flow analysis.
- *Code Transformation*: Once the test-cases are generated, we will replace the call of source-missing library functions by appropriate concrete values produced as if the functions are actually called.
- *Model-oriented Translation*: It will translate the self-complete programs into model descriptions, on which a model checker will perform a proper verification to find the real bugs if occurring.

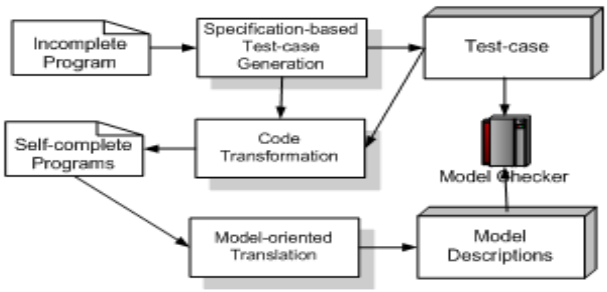

Figure 1. The specification-based model checking framework

V. EFFICIENT CONSTRAIN-BASED TEST-CASE GENERATION

When we combine the constraints generated from the function specification and the program flow, it will reach the exponential complexity since all of possible combinations of constraints must be explored.

To reduce the complexity, we then introduce an algorithm named CTG^E (Efficient Constraint-based Test-cases Generation), which is shown in Fig. 2. CTG^E aims at producing test-case from combination of two sets of constrains. The main idea of CTG^E is that it does not try to make all possible combined constraints. Instead, CTG^E processes each constraint of a certain set. For each path condition, CTG^E first produces an appropriate test-case. Then, it calls a sub-procedure named *combine* to further process.

[1] The parameters of *n* in the pre/post condition will be replaced by *a* in the constraints performing the inter-procedure analysis
[2] There would be $2 \times 4 = 8$ possible combination of constraints, among which only 4 are satisfiable. In Section V we will discuss how to reduce the complexity of constraint combination

DOI: 01.IJIT.02.02.46

For every test-case t processed in *combine*, a specific function named *symbolic_exec* will be called to find the constraints which t belongs to. The operation of *symbolic_exec* will perform *symbolic execution*, a classical technique to trace the execution path of given input by tracking symbolic rather than actual values. Based on that, *combine* keeps generating relevant constraints and calls itself recursively to generate more suitable test-cases. During the whole process of CTG^E, we also make use of a special constraint named C_{mark} which marks the explored parts in the space of test-case domain. Therefore, CTG^E can avoid duplication when generating constraints and test-cases.

For example, let us consider the two following sets of test-case $P = \{P_1=(n>0);\ P_2=\neg(n>0)\}$ and $Q = \{Q_1=n>3;\ Q_2=\neg(n>3)\}$. First, *CTG^E* generates randomly a test-case for a constraint. Let it be n=4 for P_1. Performing symbolic execution on the test-case, one can realize that the test-case falls into the combined constraint

Algorithm: CTG^E (Efficient Constraint-based Test-cases Generation)
Input: V_P, V_E: two set of path constraints
Output: T : set of test-cases
Operations
$T = \emptyset$
$C_{mark} = \emptyset$
Foreach (path constraint $\chi \in V_P$)
  t = *solve_constraint* ($\chi \cap \neg C_{mark}$)
  combine(t)
End For

SubProcedure *combine* (test-case t)
Begin
  add t to T
  α = *symbolic_exec*(t, V_P)
  β = *symbolic_exec*(t, V_E)
  $C_{mark} = C_{mark} \cup (\alpha \cap \beta)$
  if ($\alpha \cap \neg\beta \cap \neg C_{mark}$) $\neq \emptyset$ **then**
    combine(*solve_constraint*($\alpha \cap \neg\beta \cap \neg C_{mark}$))
  end if
  if ($\neg\alpha \cap \beta \cap \neg C_{mark}$) $\neq \emptyset$ **then**
    combine(*solve_constraint*($\neg\alpha \cap \beta \cap \neg C_{mark}$))
  end if
End

Figure 2. Efficient Constraint-based Test-case Generation (CTG^E) algorithm

$P_1 \wedge Q_1 = n>0$ && $n>3 = n>3$. Then, *CTG^E* tries to solve the formula $P_1 \wedge \neg Q_1 \wedge \neg C_{mark}$ with C_{mark} being updated as $C_{mark} = P_1 \wedge Q_1$. We have $P_1 \wedge \neg Q_1 \wedge C_{mark} = n>0$ && $\neg(n>3)$ && $\neg(n>3) = n>0$ && $n \leq 3$. Then, a test-case is generated accordingly, e.g. n = 2.

Next, *combine*(2) is invoked, which is corresponding to the constraint $P_1 \wedge Q_2$ with C_{mark} being updated as $n>3 \cup n>0$ && $n \leq 3 = n>0$. We then have $P_1 \wedge \neg Q_2 \wedge \neg C_{mark} = n>0$ && $n>3$ && $!(n>0) = \emptyset$, then this formula is not considered.

Meanwhile, we have $\neg P_1 \wedge Q_2 \wedge \neg C_{mark}$ = $!(n>0)$&&$!(n>3)$&&$!(n>0) = nd$"0. Solving this constraint, one, for instance, gets a new test-case of n=-7. Then, *combine(-7)* is invoked accordingly. At the moment, C_{mark} is updated as $n>0 \cup !(n>0)$ && $!(n>3) = n>0 \cup n \leq 0$, making $P_2 \wedge \neg Q_2 \wedge \neg C_{mark} = \neg P_2 \wedge Q_2 \wedge \neg C_{mark} = \neg P_1 \wedge Q_1 \wedge \neg C_{mark} = \emptyset$. Thus, the algorithm stops with no more test-cases generated.

Theorem 1. *The set of test-cases generated by CTG^E algorithm is sufficient to cover all of possible valid combined constraints.*

Proof. Assuming that an undirected graph $G = \langle V,E \rangle$ constructed as follows: each vertice v in V corresponds to a solvable combined constraint generated by the CTG^E algorithm. An edge $e_{ij} = (v_i, v_j)$ is added to E if v_i and v_j are sub-constraints of a constraint in either V_P or V_N.

For example, in Fig. 3 is the graph constructed when we consider the set constraints P and Q previously discussed. In the graph, there are three vertices corresponding to three solvable constraints. There is an edge connecting v_1 and v_2 since their constraints are both sub-conditions of P_1. Similarly, v_2 and v_3 are connected since their constraints are both sub-conditions of Q_2

considered visited if *CTG^E* produces a test-case satisfying the corresponding combined constraint of v. If all vertices in G are visited after *CTG^E* finishes, then *CTG^E* generates sufficient test-cases to cover all of possible valid combined constrained.

When *CTG^E* begins, it starts by a certain test-case I generated to satisfy a path constraint α of V_P. Using symbolic execution, one can determine the path condition β of V_N which I belongs to. It means that a vertice $q = \alpha \cap \beta$ just has been initially visited.

Consider the formula $\alpha \cap \neg\beta$ referring to a vertice q', which should be connected to q since $\alpha \cap \beta$ and $\alpha \cap \neg\beta$ are both sub-constraints of α. Let C_{mark} be the formula

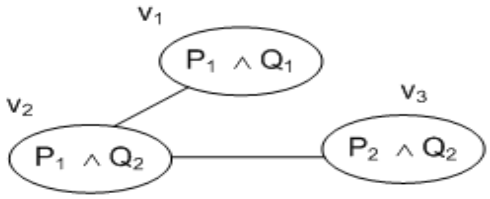

Figure 3. A graph representation of combined constraints

representing all of vertices already visited (i.e. the combined constraints whose corresponding test-cases have been generated already). Similarly reasoning, we finally obtain that the two formulas $\alpha \cap \neg\beta \cap \neg C_{mark}$ and $\neg\alpha \cap \beta \cap \neg C_{mark}$ should represent all vertices connecting to q which have not been visited. By recursively solving those formulas and updating C_{mark} in the sub-procedure *combine*, *CTG^E* will iteratively visit all of vertices in the connected component which q belongs to.

Lastly, one can note that by checking all of constraints of V_P, *CTG^E* will travel to all possible connected components of G. Thus, all vertices of G will be logically visited when *CTG^E* performed and there are no vertices doubly visited. □

Complexity Analysis. Performing elementary analysis on *CTG^E*, one can realize that *CTG^E* will involve the embedded solver $2K$ times, with K is the number of test-cases generated and $K \leq N+M$ where N and M are the path constraints on V_P and V_E respectively. If we take into account the actions of generating N path conditions on V_P, the total complexity of *CTG^E* will be $O(2K+M) \sim O(3N+M)$ which should be improved significantly compared to that of the original *CTG*.

DOI: 01.IJIT.02.02.46

VI. Experiments

The approach in this paper was tested in a practical situation of evaluating programming exercises of university students. The data set is collected from the programming work submitted by students from Faculty of Computer Science and Engineering, Ho Chi Minh City University of Technology. The requirements to be fulfilled in this experiment are non-trivial programming problems given to students. The list of problems is given in Table 1, which also gives the information of the combined constraints make from path conditions. For loop-based programs, the path constraints are computed using the coverage analysis technique [8], in which the loops are enforced to repeat respectively 0, 1, 2 and more than 2 times. Thus, the algorithm may have some limitations on programs with complicated loops.

The dataset used in this experiment is collected from the work of 50 students. In fact, there are actual marked programming works. Basically, for each programming problem, we annotate the student works to get their marked automatically using some verification tools. However, as students recently have been allowed to use library functions, e.g. mathematical functions defined in <*math.h*>, our existing approach purely relying in model checking is hindered significantly. With the proposed framework in this paper, we can now evaluate student work in an automatic manner.

We also do compare the performance of our approach with the typical white-box approach. When manually inspecting, it was observed that there are only 89% students' bugs detected using white-box approach. Exact information on improvement of bug detection is given in Table II. When the constraint-based approach is applied with teachers' sample solutions playing the roles of original versions and student works evolved versions, the performance of bug detection is significantly improve with 98% bugs detected. Few bugs are still missed because our solver fails to resolve some complex non-linear expression in path conditions.

TABLE I.
Programming Problems Used as Experimental Data

No	Problem	Constraint	Solver calls (brute-force)	Solver calls (CTG^E)
1	Leap year checking	14	42	40
2	Triangle classification	22	89	31
3	Date validation checking	62	736	90
4	Time validation checking	28	96	37
5	Factorial computing	28	96	58
6		28	96	56
7	Prime number checking	56	384	92
8	Sum of 1..*n*	25	84	54

TABLE II.
Bugs Detected by Brute-force and Combined Constraints Approach

Problem No	Real Bugs	Detected by white-box	Detected by $CTG^{(E)}$
1	12	11	12
2	10	6	10
3	12	10	10
4	13	10	13
5	14	14	13
6	11	11	11
7	12	12	12
8	12	12	11
Total	96	86(89%)	94(98%)

VII. Conclusions

In this paper, we present an approach to verify incomplete programs, which reflect a practical situation that the source code of whole software project may not be always available. The approach is based on the concolic technique. In particular, to tackle the problem of analyzing source-missing functions, we propose to combine the functions specification with the program flow. Thus, we can still generate sufficient test-case covering all of real execution scenario.

Our approach has been applied in a practical application of checking programming works of students, where the code submitted by students often involving source-missing library functions. Experimental results showed that this approach has gained some initial promising results.

Acknowledgment

This work is part of the Higher Education Project 2 project (supported by World Bank and Hochiminh – Vietnam National University).

References

[1] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching-time", in *Lecture Notes in Computer Science*, vol. 131, Springer-Verlag, 1981.

[2] D. A. Duffy, *Principles of Automated Theorem Proving*, John Wiley & Sons, 1991.

[3] P. Godefroid, N. Klarlund, K. Sen, "DART: directed automated random testing", in *Programming Language Design and Implementation*, pp. 213-223. ACM, 2005.

[4] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, S. K. Rajamani, Synergy: "A new algorithm for property checking", in *FSE '06: Foundations of Software Engineering (ACM SIGSOFT Distinguished Paper)* , 2006.

[5] C. Popeea, W. N. Chin, "Dual analysis for proving safety and finding bugs", in *Proceedings of the 2010 ACM Symposium on Applied Computing*, pp. 2137-2143, Switzerland, 2010.

[6] N. E. Beckman, A. V. Nori, S. K. Rajamani and R.J.Simmons, "Proofs from tests", in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, New York, USA, ACM Publisher, 2008.

[7] Sen, K., Marinov, D., and Agha, G., "CUTE: a concolic unit testing engine for C", in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT international Symposium on Foundations of Software Engineering (Lisbon, Portugal, September 05 - 09, 2005). ESEC/FSE-13*. ACM, New York, NY, 263-272, 2005.

DOI: 01.IJIT.02.02.46

[8] Andreas Spillner, Tilo Linz, Hans Schäfer, *Software Testing Foundations - A Study Guide for the Certified Tester Exam - Foundation Level - ISTQB compliant*, 1st print. dpunkt.verlag GmbH, Heidelberg, Germany, 2006.

[9] S. Kripke, "Semantical considerations on modal logic", *Acta Philosophica Fennica*, 16:83–94, 1963.

[10] Venema, Yde, "Temporal logic", in *Goble, Lou, ed., The Blackwell Guide to Philosophical Logic*, Blackwell, 2001.

[11] G. Pace, N. Halbwachs and P. Raymond, "Counter-example generation in symbolic abstract model-checking", *International Journal on Software Tools for Technology Transfer (STTT)*, 5(2): 158 – 164, 2004.

[12] J. Zhang, Z. Huang, Z. Cao and F. Xiao, "Counterexample generation for probabilistic timed automata model checking", In *Proceedings of International Conference on Computer Science and Software Engineering*, China, pp. 210-214, 2008.

[13] T. Han, J.P. Katoen and B. Damman, "Counterexample generation in probabilistic model checking", *IEEE Transactions on Software Engineering*, 35(2): 241-257, 2009.

[14] M. Chechik and A. Gurfinkel1, "A framework for counterexample generation and exploration", chapter in *Fundamental Approaches to Software Engineering*: 220-236, Springer, 2005.

[15] P. Gastin and P. Moro, "A framework for Counterexample Generation and Exploration", chapter in *Model Checking Software*: 24-38, Springer, 2007.

DOI: 01.IJIT.02.02.46